AFRL-IF-RS-TR-2002-75
Final Technical Report
April 2002

COMPUTER SECURITY ASSISTANCE PROGRAM FOR THE TWENTY-FIRST CENTURY (CSAP21) ADVANCEMENT AND EXPERT TECHNOLOGY EXCHANGE (CAETE)

WetStone Technologies

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-75 has been reviewed and is approved for publication.

APPROVED:

JAMES L. SIDORAN
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	APRIL 2002	Final Jun 99 – Jun 00

4. TITLE AND SUBTITLE
COMPUTER SECURITY ASSISTANCE PROGRAM FOR THE TWENTY-FIRST CENTURY (CSAP21) ADVANCEMENT AND EXPERT TECHNOLOGY EXCHANGE (CAETE)

5. FUNDING NUMBERS
C - F30602-99-C-0041
PE - 33140F
PR -: 7920
TA - 09
WU - P2

6. AUTHOR(S)
Chester Hosmer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
WetStone Technologies, Incorporated
17 Main Street, Suite 237
Cortland New York 13045

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFGB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2002-75

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: James L. Sidoran/IFGB/(315) 330-3174/James.Sidoran@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*
This contract final technical report documents the CSAP21 Advancement and Expert Technology Exchange (CAETE) project results. This project expanded the capabilities of the Computer Security Assistance Program for the Twenty-First Century (CSAP21) system of systems architecture by enhancing and expanding the functionality of the Network Monitoring and Assessment (NMA) module of the Interactive Information Protection Decision Support System (IIPDSS) testbed.

14. SUBJECT TERMS
Network Monitoring and Assessment, Intrusion Detection, CSAP21

15. NUMBER OF PAGES
42

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

List of Tables

1 Period of Performance

This report reflects performance from 3/1/99 – 12/31/99.

2 Detailed Program Schedule

The following represents the current detailed program schedule.

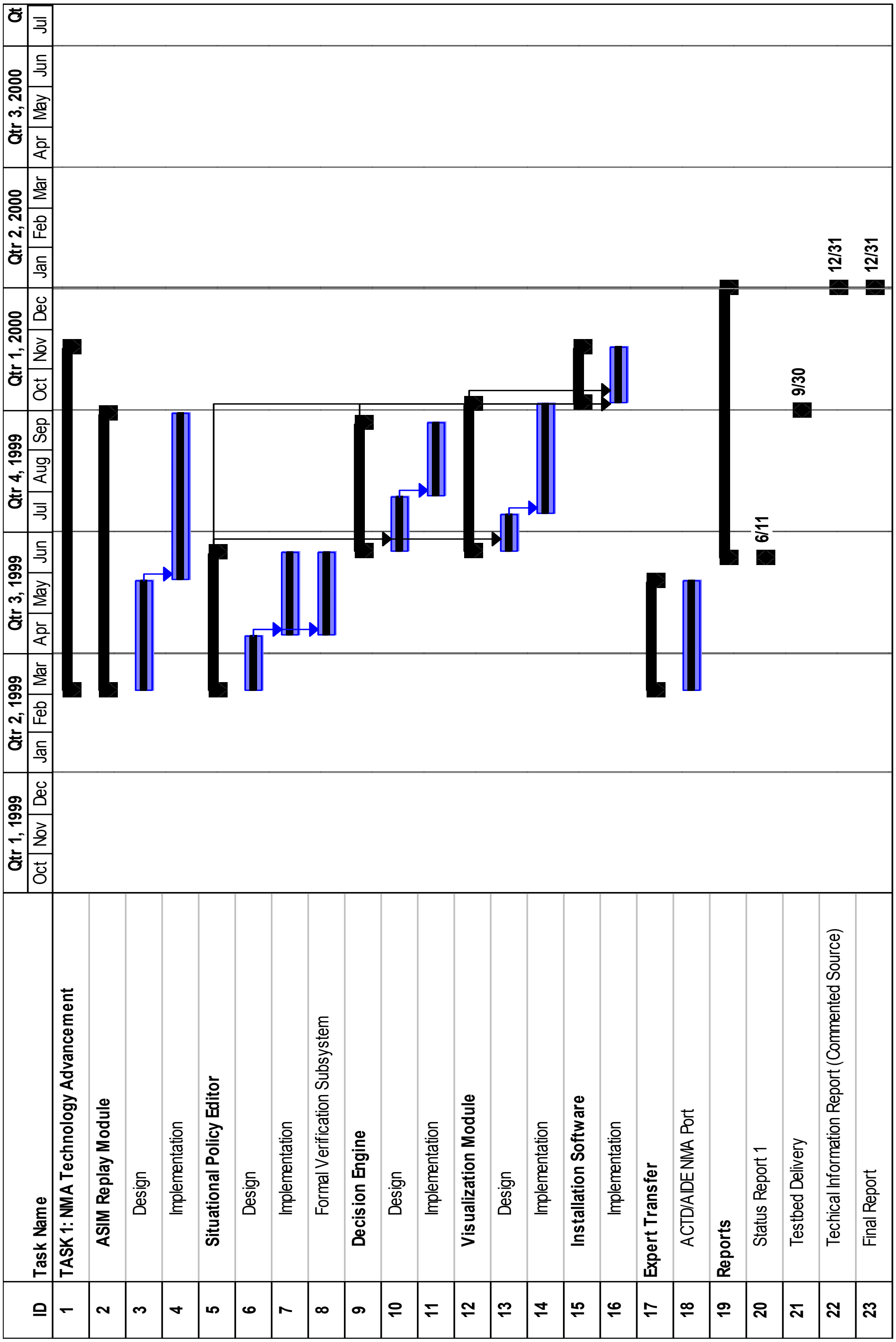


Figure 1 - Schedule

3 Background

WetStone Technologies is advancing the Computer Security Assistance Program For the Twenty-First Century (CSAP21) System of systems architecture by moving the Network Monitoring and Assessment (NMA) module forward to the point of installing it at several key components at U.S. Air Force installations. These installations will promote furthering our communications with analysts in the trenches, thus allowing us to further enhance the existing NMA and Visualization Component (VC).

WetStone Technologies has recently completed an effort designed to advance the state-of-the-art in information protection by rapidly moving key aspects of the CSAP21 system of systems architecture forward. The CSAP21 Module Demonstration project (Contract #: F30602-98-C-0220) has produced a proof of concept demonstration that provides key capabilities that can be used immediately by the U.S. Air Force to improve the information security posture.

3.1 Major Accomplishments:

The following table represents the original requirements and the refinements made during the effort.

Original Requirement	Refinement
Develop a software bridge to ASIM 2.0 allowing for a real-time connection and processing of ASIM reports.	Developed a software bridge to provide replay of ASIM 1.7 and 2.0 log files. This requirement was modified from the original because we were not allowed to instrument (add software to) the ASIM hardware platform. Instead, we developed a GUI application that allows the simultaneous replay of multiple ASIM log files (version 2.0 and 1.7)
Formalize the NMA Knowledge Base in order to prove the consistency of the rules table.	Completed as specified
Expand the fuzzy data fields evaluated by the NMA during it decision making phase. The contractor will examine additional data from ASIM such as key word violations and service operations as possible extensions of the table and fuzzy sets.	Completed as specified
Not Specified	Advanced the fuzzy knowledge base concept to include the ability to develop situational decision making policies.
Work on-site with the AFRL staff at Rome and their contractors to advance information protection capabilities into adjunct projects such as EPIC and ACTD/AIDE	Ported the Network Monitoring and Assessment Module (NMA) and Policy Editor to Solaris and integrated these technologies with the AIDE environment.
Port the NMA, VC and the ASIM 2.0 bridge software to a PC notebook computer for inclusion into the IIPDSS testbed.	Completed as specified
Install the NMA and Visualization Component (VC) modules at 3 installations in order to receive feedback from analysts on their operation and use. Perform system upgrade per user feedback	Produced an installable CD that has been distributed to many locations for installation and experimentation. The current installations include. DARPA TIC, DISA JPO, AFRL-HQ, AFRL-HE, AFIWC, AFRL Rome Cyber-Forensics Lab, and HQ AFSPC/SC.

Table 1 - Accomplishments

4 Project Activity

4.1 Travel

Date	Destination	Purpose of Trip
3/8/99	AFRL – Rome, NY	Planning Meetings
3/19/99	AFRL – Rome, NY	Initial demonstration
3/24/99	AFRL – Rome, NY	Planning Meetings
4/8/99 – 4/13/99	WetStone NY Offices	Formal Kickoff Meeting
4/12/99 – 4/17/99	AFRL – Rome, NY	AIDE Installation
4/21/99	AFRL – Rome, NY	Meetings
5/10/99 – 5/11/99	AFWIC – San Antonio, TX	Demonstration & Meetings
5/12/99 – 5/13/99	Ithaca, NY	Conference & Meetings
5/14/99	CFRDC – New Hartford, NY	Meetings & Planning
5/20/99 – 5/21/99	AFRL – Rome, NY	Software Development
5/25/99	HQ – Dayton, OH	Demonstration & Meetings
6/9/99	AFRL, Rome, NY	Software Support
9/17/99	AFRL, Rome, NY	Demo NET-FLARE
9/28/99	AFRL, Rome, NY	Software Delivery

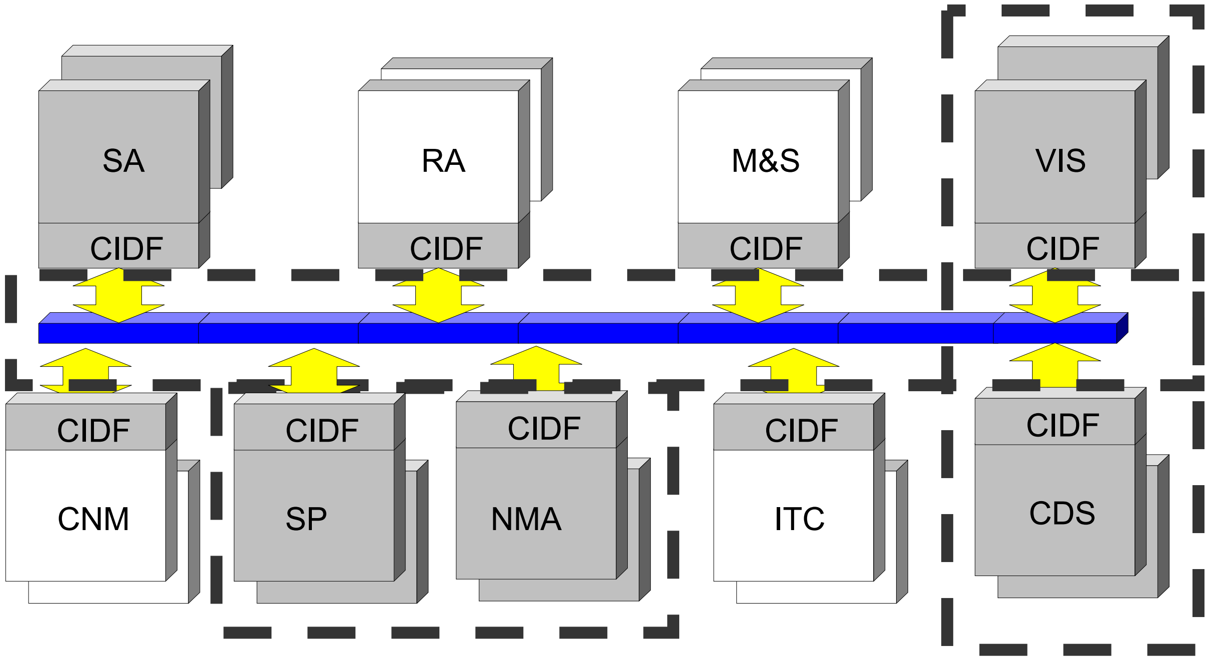
Table 2 – Travel

4.2 CAETE Architecture

The CAETE architecture is based on several simple concepts and is depicted below.

1. System components communicate over an open bus architecture
2. Multiple components of each type are encouraged
3. The bus structured is a simple TCP/IP messaging based infrastructure that carriers CIDF messages.
4. Visualization is provided as a separate and distinct component. The Visualization Component provides decision support and course of action recommendations to the information warrior.

Based on this simple architecture during this effort we instantiated several key components of the architecture and demonstrated their interoperability and flexibility. Components include the NMA (Decision Engine), the Visualization Engine, and the Situational Policy Editor. A detailed layout of their interaction is depicted below.



FY98 Prototype Demonstration

NMA - Network Monitoring & Assessment
SP – Situational Policy
SA - Situation Assessment
CIDF - Common Intrusion Detection Format
CDS - Common Data Storage
VIS - Visualization Module

Future Implementation

ITC - Intelligence and Threat Correlation
CNM- Network Mapping
RA - Risk Assessment
M&S - Modeling and Simulation

Figure 2 - CAETE Architecture

In order to experiment and test this proof of concept prototype, we designed, developed, and tested several key re-useable software components. The table on the following page defines those components, and sections following describe the capabilities and functionality of each.

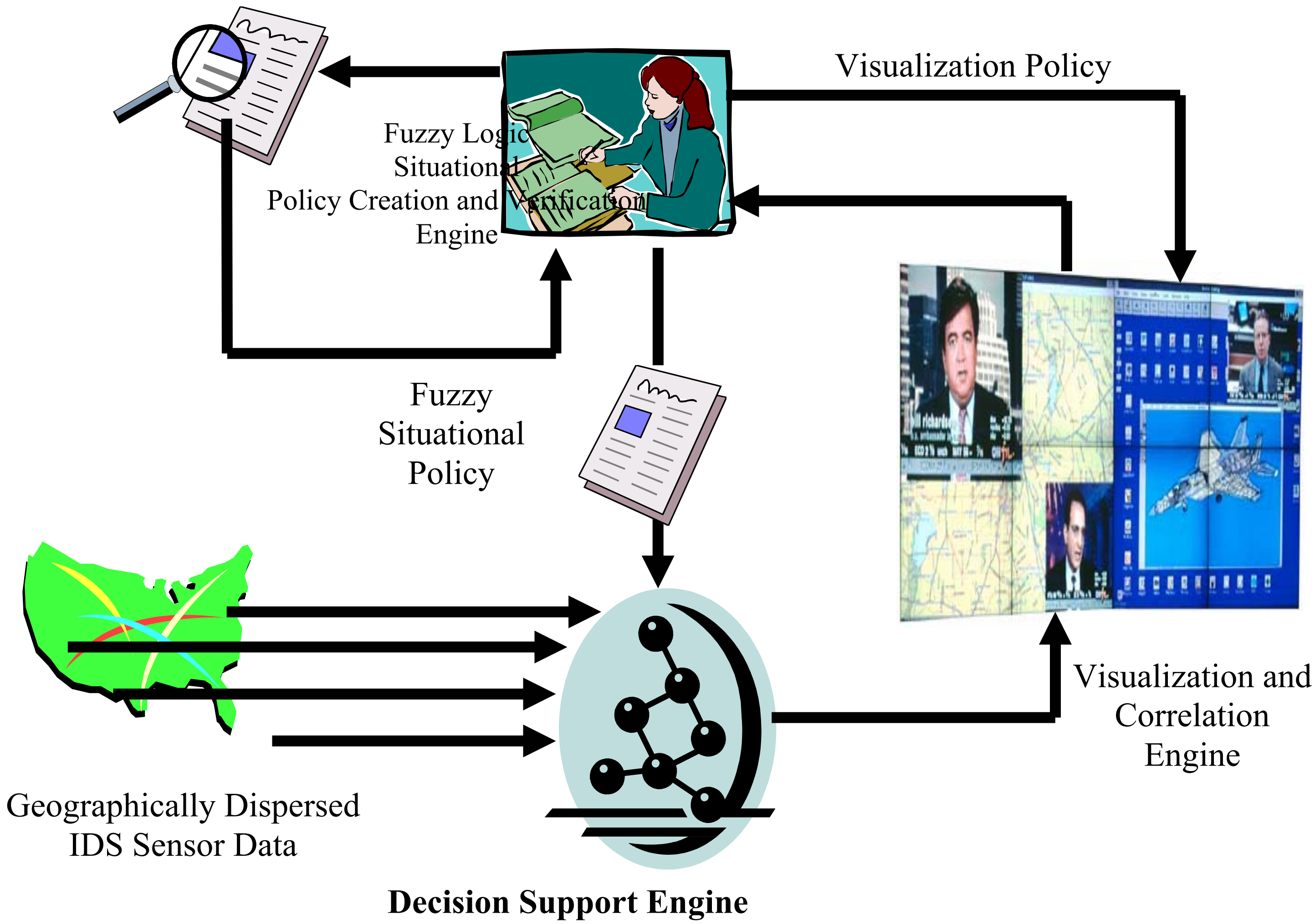


Figure 3 – Module Interaction

4.3 Completed Software Modules

Software Component	Description	Version	Source	OS
ASIM Bridge Software	Provides multiple simultaneous replay of ASIM 1.7 and 2.0 log files. The log files are communicated over the CAETE Architecture via the common data bus TCP/IP	1.1	Object Pascal	Windows NT
Situational Policy Editor NT	Provides the user with the capability of defining and creating situational decision support polices for ASIM data streams on Windows NT platforms.	1.1	Object Pascal	Windows NT
Situational Policy Editor Solaris	Provides the user with the capability of defining and creating situational decision support polices for ASIM data streams on Solaris platforms.	1.0	Java	Solaris
Network Monitoring and Assessment (NMA) Decision Engine	Receives ASIM events from the ASIM bridge and provides recommended course of actions defined by the Situational policy. The decision engine derives additional data from the ASIM events in both static and state based forms. The NMA outputs the raw data, derived data, fuzzy conversion, and recommended course of actions, classification and impact of the event. The NMA provides output via the CAETE Architecture via the common data bus TCP/IP	1.1	C++	Windows NT
Network Monitoring and Assessment (NMA) Decision Engine	Receives ASIM events from the ASIM bridge and provides recommended course of actions defined by the Situational policy. The decision engine derives additional data from the ASIM events in both static and state based forms. The NMA outputs the raw data, derived data, fuzzy conversion, and recommended course of actions, and classification of the event. The NMA provides output via the CAETE Architecture via the common data bus TCP/IP	1.0	C++	Solaris
Visualization Engine	This component receives data from the NMA from the CAETE Architecture via the common data bus TCP/IP. The input provided is the NMA raw data, derived data, fuzzy conversion and recommended course of action.	1.1	Object Pascal	Windows NT

Table 3 - Completed Software Modules

4.4 ASIM Bridge

The ASIM Bridge software application depicted below allows for the selection of ASIM log files version 1.7 or 2.0 for "accelerated time" replay. The ASIM events are replayed and fed into the CAETE architecture over the TCP/IP data bus. Any number of log files can be simultaneously replayed simulating multiple geographic sensors supply data over the CAETE infrastructure.

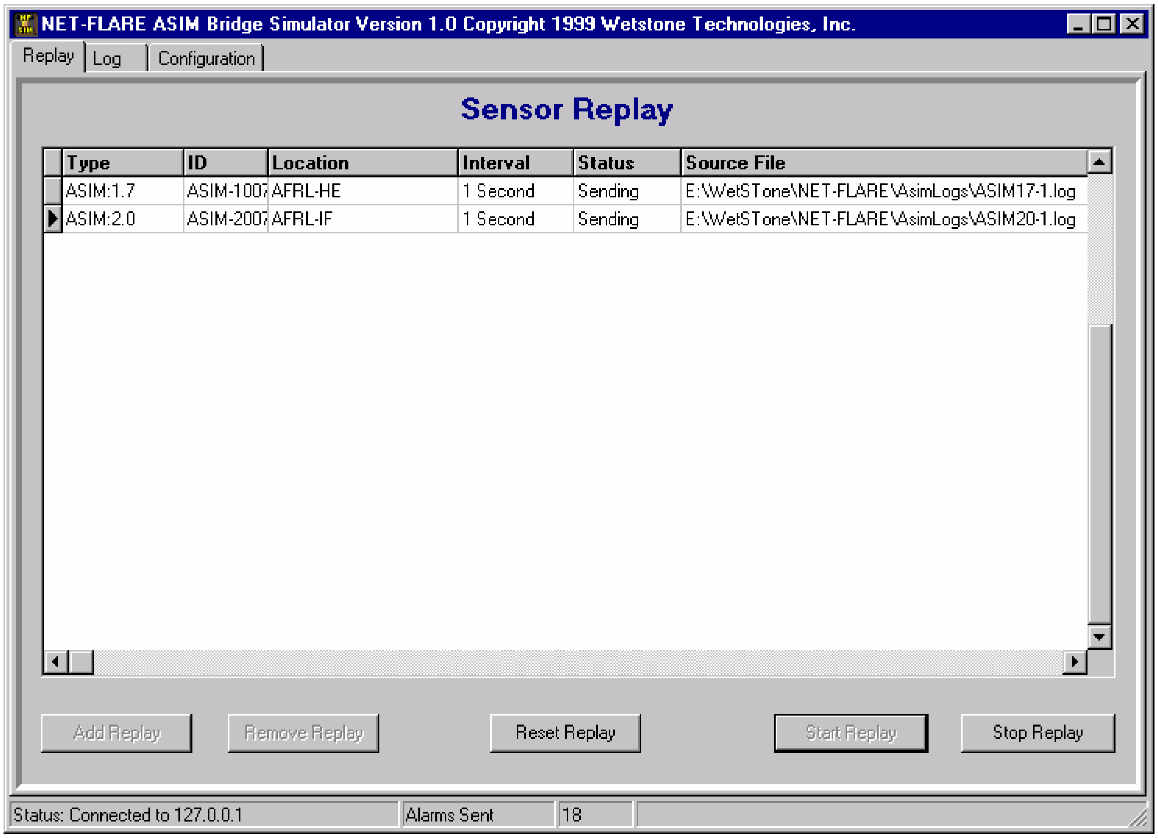


Figure 4- ASIM Bridge

4.5 Situational Policy Editor

This component provides the analyst with the ability to create, verify, and then deploy operational decision support policies that are based on a particular situation or mission. The engine tools allow the analyst to define the "meaning" or translation of raw intrusion detection sensor data values into a "set" of fuzzy values for a given situation. For most Network based IDS sensors, the type, or "kind-of" data that is provided is similar. Common data included is the time of the network session traffic, originating IP address, destination IP address, session volume, session duration, TCP/IP or UDP service type, etc. From this information, we can define a policy for conversion of this raw sensor data into meaningful fuzzy sets.

The following table provides an actual example of raw sensor data that has been converted into a set of fuzzy values based on a situational policy. You will notice that we have significantly expanded the information available from the sensor into a rich set of fuzzy values.

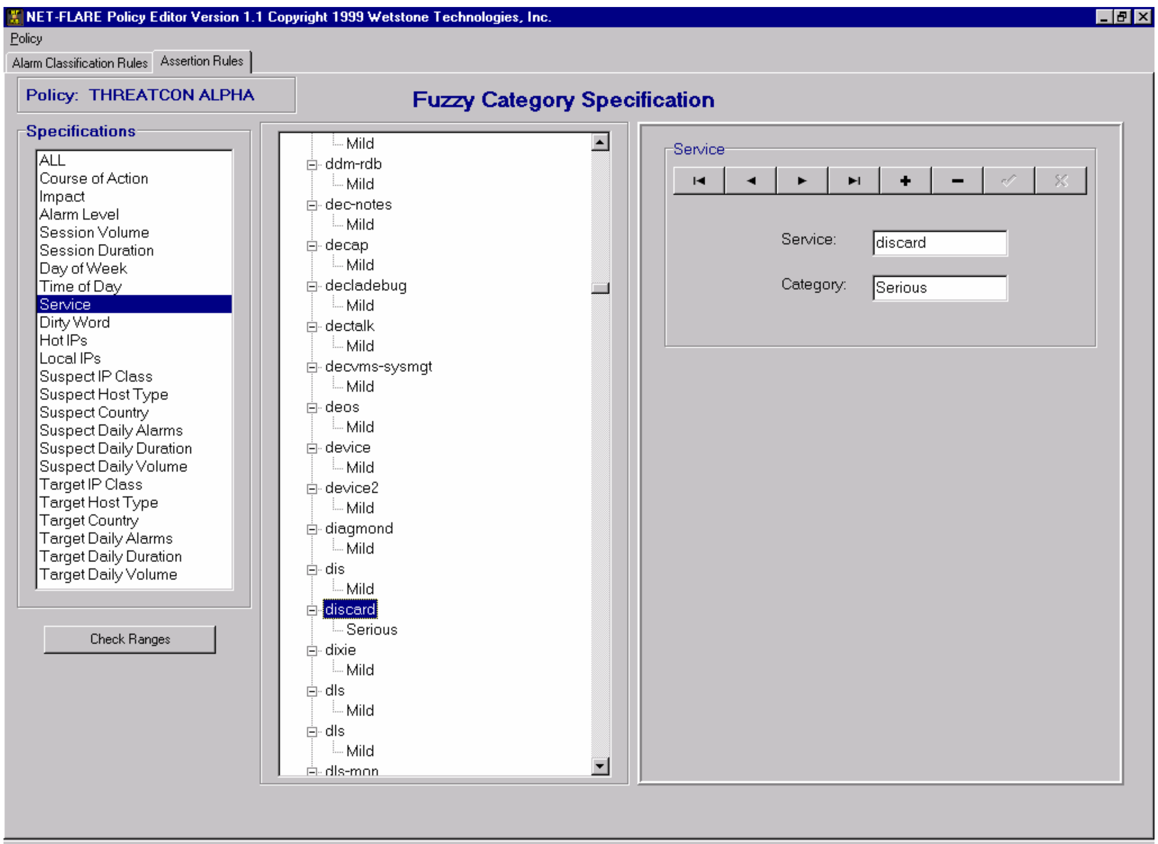


Figure 5 - Situational Policy Editor

Based on this set of possible fuzzy values, the analyst can construct rules based on the fuzzy set. Each rule combines the possible fuzzy values together and then determines the outcome if the rule is evaluated to true. Our current policy model supports the dyadic operators of AND and OR for rule creation along with "*" or wild card support within rules. Our decision to limit the number of dyadic operators was only to simplify rule construction and verification by the analyst.

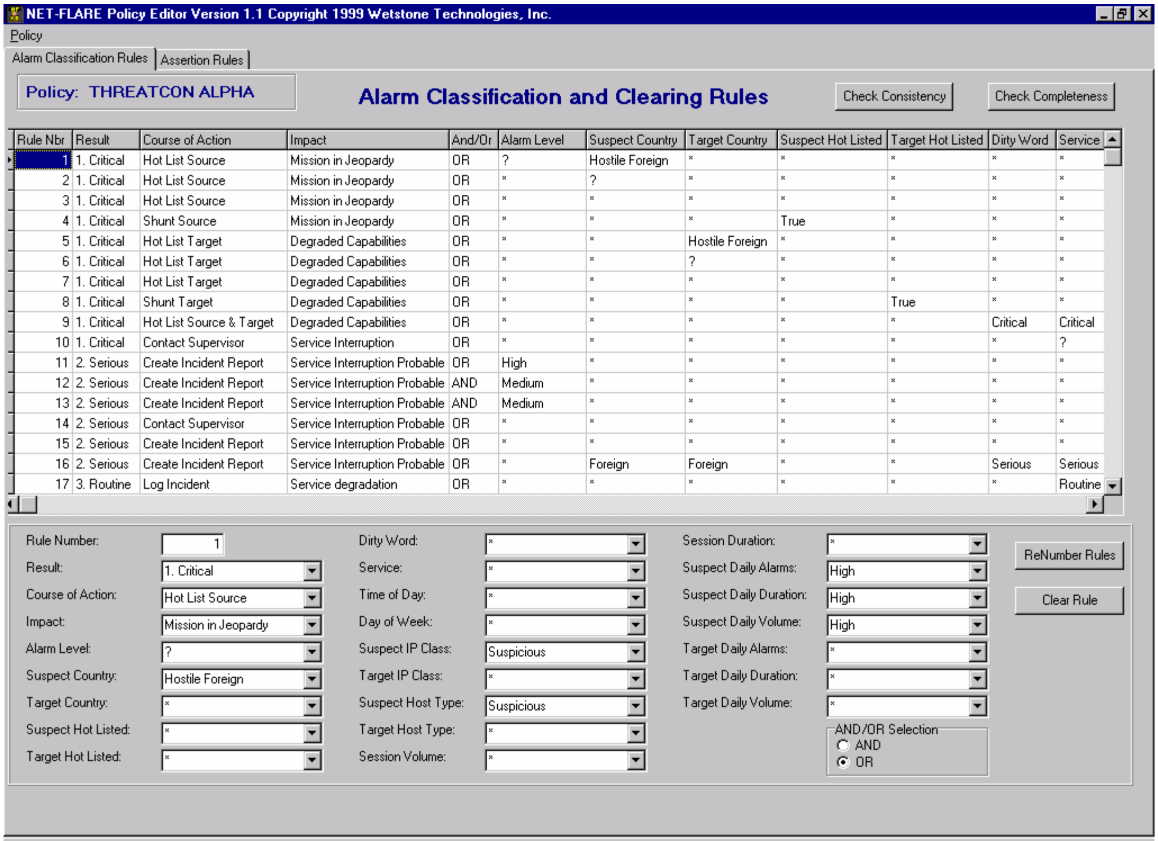


Figure 6 - Alarm Classification Rules

4.6 Formal Policy Verification

The fuzzy model for rule development provides us with great flexibility for rule specification as with other expert system approaches. Major questions surrounding the use of AI technologies in decision support applications revolve around the question of accurately modeling the decision process. Verification of these models is necessary in order to ensure the accuracy and predictability of the results. In our approach it is very important that the policy accurately matches the analyst intent since the analyst in the field must be able to rapidly develop and deploy situational policies. We would like to be assured that the fuzzy policies that we create are complete – all possible combinations of data are covered, (completeness check); and that no two rules cover the same combination of data but rather assign a different course of action recommendation (consistency check).

Based on our approach, rules tables can grow to be large and it is difficult to inspect them by hand. If each data field *X* has X_i possible fuzzy values, and there are *n* fields, the total possible number of combinations is $X_1 * X_2 * X_3 \ldots\ldots * X_n$. As an example, in the ASIM rules table where n = 22, and each field has no more than 4 possible fuzzy values, the total number of combinations is greater than 10^9. Based on this, it is obviously not feasible to inspect the rules for consistency and completeness by hand.

Using techniques borrowed from Formal Methods or proofs, we have applied techniques to the matrix of rules, to confirm in a formal deterministic fashion, that we have achieved both consistency and completeness. Furthermore, these methods provide results that allow us to present the analyst with the consistency and completeness problems found, allowing the analyst to correct the conflicts.

4.6.1 Consistency & Completeness Issues

Verify the consistency and completeness of a table with *n* columns and *m* rows. Each row represents a “rule”. Each rule consists of assigned values for each fuzzy category as well as the recommendation for this particular combination of values. A user can change the number of columns and rows arbitrarily, (i.e. add new rules and categories). We assume the following definitions:

Consistency: there are no rows covering the same rule but stating different recommendations.
Completeness: all possible combinations of fuzzy category values are taken into account.

Recommendation	Fuzzy Category 1	Fuzzy Category 2	….	Fuzzy Category n
R	V1	V2	…	Vn
R	…	…	…	Vn
R	V1	V2	…	Vn

Table 4 - Consistency & Completeness

Possible Values:

$R = \{r_1, r_2, \ldots., r_k\}$
V1 = {v11, v12, …, v1,num_values[cat1] }, or V1= *, which means "any of the possible values".
…
Vn = {vn1, vn2, …, vn, num_values[catn]}, or Vn = *, which means "any of the possible values".

m, n, k, num_values[cat1],… , num_values[catn], are finite integers.

4.6.2 Solution

Use a software tool with the following constraints:

Input:

- names of fuzzy categories and all their possible values
- all possible values for recommendation
- contents of the table

Output:

- boolean output stating if consistency is satisfied, and if not, which rules are conflicting
- boolean output stating if completeness is satisfied, and if not, which rules are missing

4.6.2.1 Consistency

Consistency can be checked relatively easily. Assuming that each row is of the form:

Ri Vi1 Vi2 Vi3 … Vin

We would check each row against all other rows to find if they cover the same rule. Rows i and j cover the same rule if:

For all columns c, c= 0, …., n:
Vic = Vjc, or Vic= * or Vjc = *, i = 0, .., m, j = 0, …m

In order to improve performance, an efficient comparison algorithm should be used. The most primitive, slowest algorithm that illustrates the issue would be:

```
for i = 0, … m
{
     rule1 = Ri Vi1 Vi2 … Vin

     for j = i+1, …, m:
     {
          rule2 = Rj1 Vj1 Vj2 … Vjn
```

```
                if (rule1 == rule2)
                {
                     if (Rj1 == Rj2) not consistent
                     else            consistent
                }
           }
      }
```

4.6.2.2 Completeness

There are two major approaches to the problem.

In the first approach, we can "blow up" all * entries in the table, eliminate duplicate rows, and count the resulting lines. The number of lines covering all possible rules must be:

num_values[cat1] * num_values[cat2] * …. * num_values[catn]

The problem with this approach is that we would not know which rules are missing.

In the next approach, we can generate a table of all possible rules, and check our table against this gold standard. This would allow us to know which rule(s) are missing. The consistency check would be obtained "for free" as we search the table.

Assume that the "gold standard table" has rows called GoldStandardRule, of the form:

Rg V1g V2g … Vng

Assume that the able to be checked has rows of the form:

R V1 V2 … Vn, i = 0, …, m

The algorithm for checking would be:

```
     //For each GoldStandardRule, search the table to find all
rows that match it:

     For all GoldStandardRules rn = 0, …,
num_values[cat1]*…*num_values[catn]:
     {
          For all rows of the table, i = 0, …., m:
          {
               For all columns of the table, c= 0, …, n:
               {
                    if (Vc = Vg, or Vc = *)       match
                    else                         no match
```

```
            }
        }
        //if no matches have been found for this
        GoldStandardRule,
        // consistency is violated


        //For all matching rules found, check if their
        recommendations are //different
    }
```

For example, if a GoldStandardRule is:

rx	a	b	c

And the table has matching entries covering this rule (assuming that each category can have a,b, or c as possible values):

rx	a	*	c
rx	*	*	c
rz	a	b	*

Then consistency would be violated for the first and second table rules, because * expanded in those rules generates rules that cover the rule a,b,c.

One of the issues with this approach is the automatic generation of all possible rules without knowing ahead of time how many categories are involved. This issue can be solved if we start from the rule with all * entries, and "blow up" the entries one by one.

The major problem with this approach is the size of the tables generated. For example, if there are 10 categories, each with 4 possible values, then there are 4^10 = 1,048,576 possible rules.

Since the table supplied for checking should have many entries with * values, it would be prudent to use this opportunity. We propose to use the following approach, based on a "tree" structure:

```
     Search the table for rule * * … *
     If not found, "blow up" the first *, (i.e. go one level
down, and search for rules:
          v11 * … *
          v12 * … * …
          …
          v1, num_values[cat1] * … *.
          For each rule not found in the table, continue to
          "blow up" the next * in the
          rule and search the table.
```

Graphically, we can picture this tree structure. Assume that we have 3 categories, each with possible values of a,b, and c. The tree would look like that presented below. The nodes are rules, and the number of branches at level *i* equals *num_values[cat i]*. The tree dynamically grows as we search the table for particular rules. Whenever we find a node of this tree present in the table, we know that all rules in that branch are covered in the table, and we need not check for their presence in the table. Therefore, we need not expand this branch of the tree any further. In the worst case, we have to generate nodes at the maximum possible depth, (i.e. generate the "golden standard" rule table), and we have to search the table for each node of the tree. Searching the table should not present a performance issue, since the table is always relatively small, perhaps in the order of 100 rows and 20 columns. In this approach, consistency checking does not come for free anymore but has to be done separately.

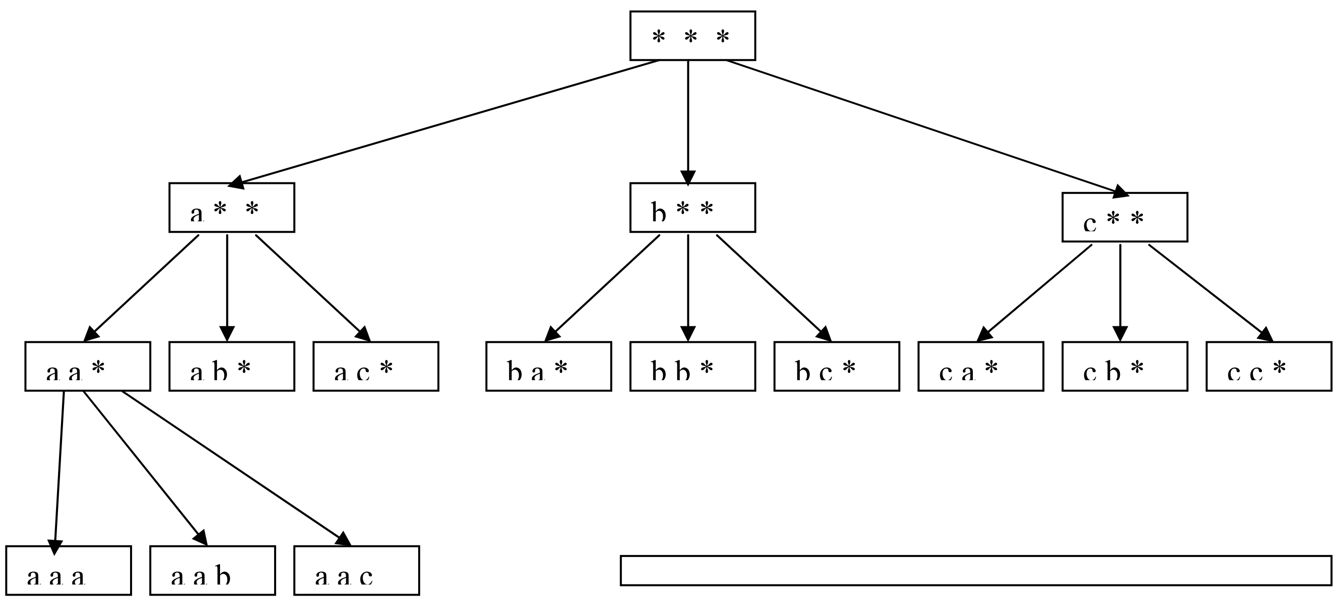


Figure 7 - Decision Tree Structure

4.6.2.3 Algorithm for Completeness Check

```
list <string>       pattern;
list <pattern>      Patterns;
list <pattern>::    iterator y;

for (ii = 0; ii < number_categories; ++ii)
{
     pattern += "*";
}

Patterns.push_back(pattern);
y = Patterns.begin();

while (y < Patterns.end())
```

```
{
     pattern = *y;
     if (table.find(pattern) != table.end())  //found pattern in
the table
     {
          if ( only_stars(pattern))
               cout << "Unsafe!\n";
          Patterns.erase(y);
     }
     else
     {
          if (pattern.find('*') != pattern.end())  //pattern
contains at least one *
          {
               BlowUp(pattern, Patterns);
               Patterns.erase(y);
          }
          else
          {
               cout << ""Rule " << pattern << "not matched.\n";
          }
     }

     if (y != Patterns.end())
     {
          y++;
     }
}

bool only_stars(list<string> pattern)
{
     bool res = true;
     list<string>::iterator  y;

     for (y = pattern.begin(); y< pattern.end(); ++y)
     {
          if (*y != '*')
               res = false;
     }
     return res;
}

void BlowUp(list <string>& pattern, list<pattern>& Patterns)
{
     list<string>::iterator fs;

     fs = pattern.find('*");
```

```
      for (ii = 0; ii < num_values(category fs); ++ii)
      {
            *fs = category fs(value ii);
            Patterns.push_back(pattern);
      }
}
```

4.6.2.4 Algorithm for Consistency Check

```
List<string> row1;
List<string> row2;
List<string>::iterator y;
List<row> table;

for (y = table.begin(); y < table.end(); ++y)
{
      row1 = *y;
      for ( x = y+1; x< table.end(); ++x)
            row2 = *x;
            if ( equal(row1.begin(), row1.end(), row2.begin(),
cover_same_rule(row1,row2))
            {
                  if (row1.front() == row2.front())
                  {
                        cout << "Inconsistency with rows << row1 << '\n'
<< row2 << '\n';
                  }
                  else
                  {
                  }
            }
            else
            {
            }
      }
}

//row1 and row2 must be of the same length
bool cover_same_rule(list<string>& row1, list<string>& row2)
{
      list<string>::iterator y;
      list<string>::iterator x;
      bool result = true;

      x = row2.begin() +1 ;
      for (y = row1.begin() + 1; y < row1.end(); ++y)
```

```
    {
        if ((*y == *x) || (*y == ‘*’) || (*x == ‘*’))
        {
        }
        else
        {
            result = false;
        }
        ++x;
    }
    return result;
}
```

4.6.3 Algorithm Results

After implementing the algorithms specified above we apply them to the situational policy models we developed. The result is specific details as to the conflict between rules and the completeness of the rule tables developed.

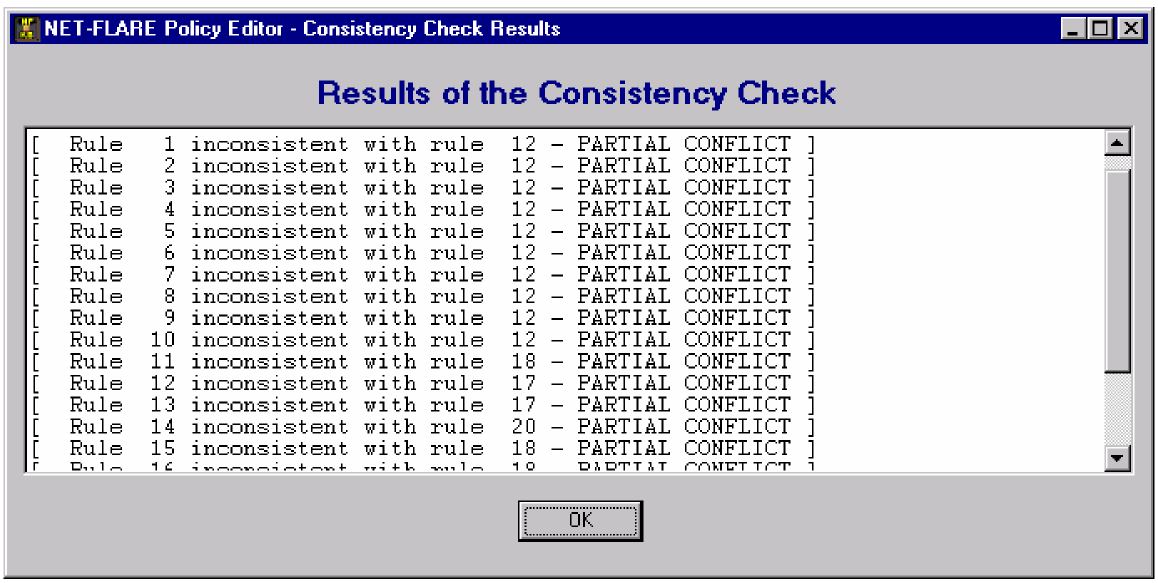


Figure 8 - Consistency Results

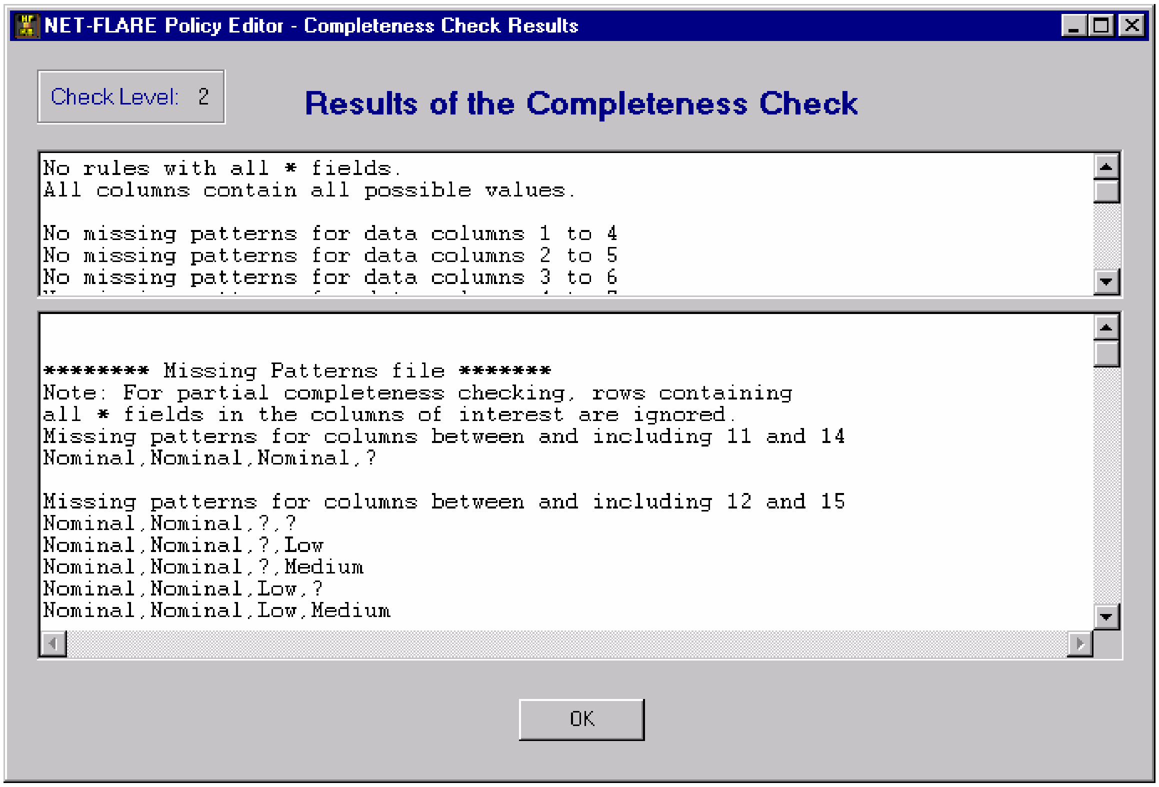


Figure 9 - Completeness Results

4.7 Network Monitoring and Assessment (NMA) Decision Engine

4.7.1 The Decision Engine

The Decision Engine (DE) is the workhorse of the decision analysis process. This program is responsible for performing the fuzzy analysis using raw sensor information based on the rules set forth by the analyst. To perform this task, the program must:

1. Parse the raw sensor data
2. Convert the raw data into a set of fuzzy values based on the currently specified policy
3. Derive additional data from the raw data (i.e. derive Day-Of-Week and Time-Of-Day from the dates and time)
4. Track state and "per-IP" statistical information (daily alarms, daily volume, daily duration)
5. Extract additional information from other sources (i.e. DNS)
6. Fire the fuzzy rules using the complete set of fuzzy values
7. Output the analysis results to the Visualization engine.

The analyst uses the Policy and Verification Engine to create and/or modify a policy. The Decision Engine communicates with the Policy and Verification Engine to obtain the current set of fuzzy values and rules to apply during the analysis process. The Decision Engine also has a Graphical User Interface that displays some state information and error messages.

The DE is a multithreaded application that can accept sensor input from multiple sources simultaneously in an asynchronous fashion. The DE's processing is simple and can analyze sensor information quickly. Operations that require more processing, such as DNS lookups, are performed in a separate thread and do not impede processing of other alarms. Additionally, in the case of DNS, network information can be cached to reduce costly DNS lookups.

The DE's innovative design is setup in such a way that the fuzzy analysis is performed independently of raw sensor data parsing. Therefore, in most cases, the DE requires little or no modification to support new sensors and new fuzzy values. One possible technological advancement of this design includes the use of a input specification language that would allow the analyst to modify the parsing rules for the sensor and thus allow any type of sensor to be added with no code modifications. This advancement is in alignment with our non-"hard-wired" approach and would extend the life cycle of this technology significantly.

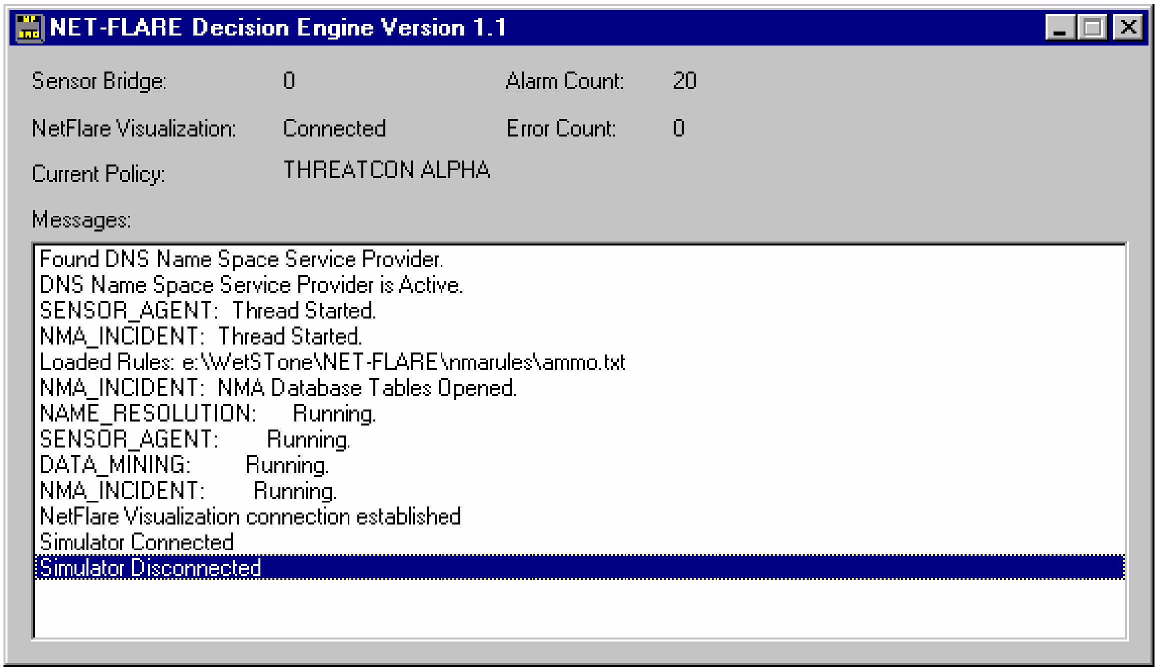


Figure 10 - NET-FLARE Decision Engine

4.8 Visualization Engine

The Visualization Engine provides the Information Warrior with an "at-a-glance" view of what has been received by the system. Fuzzy values, as well as raw values are displayed here for use, and notes may be kept on each of the active alarms in the system. The analyst has the ability to clear alarms from the system, while retaining the ability to view all data for the cleared alarms.

The analyst has the ability to generate a policy for the Visualization system giving him control of different aspects of the system. For example, colors may be selected to represent an alarm's data, based on the alarm's classification or specific fuzzy values.

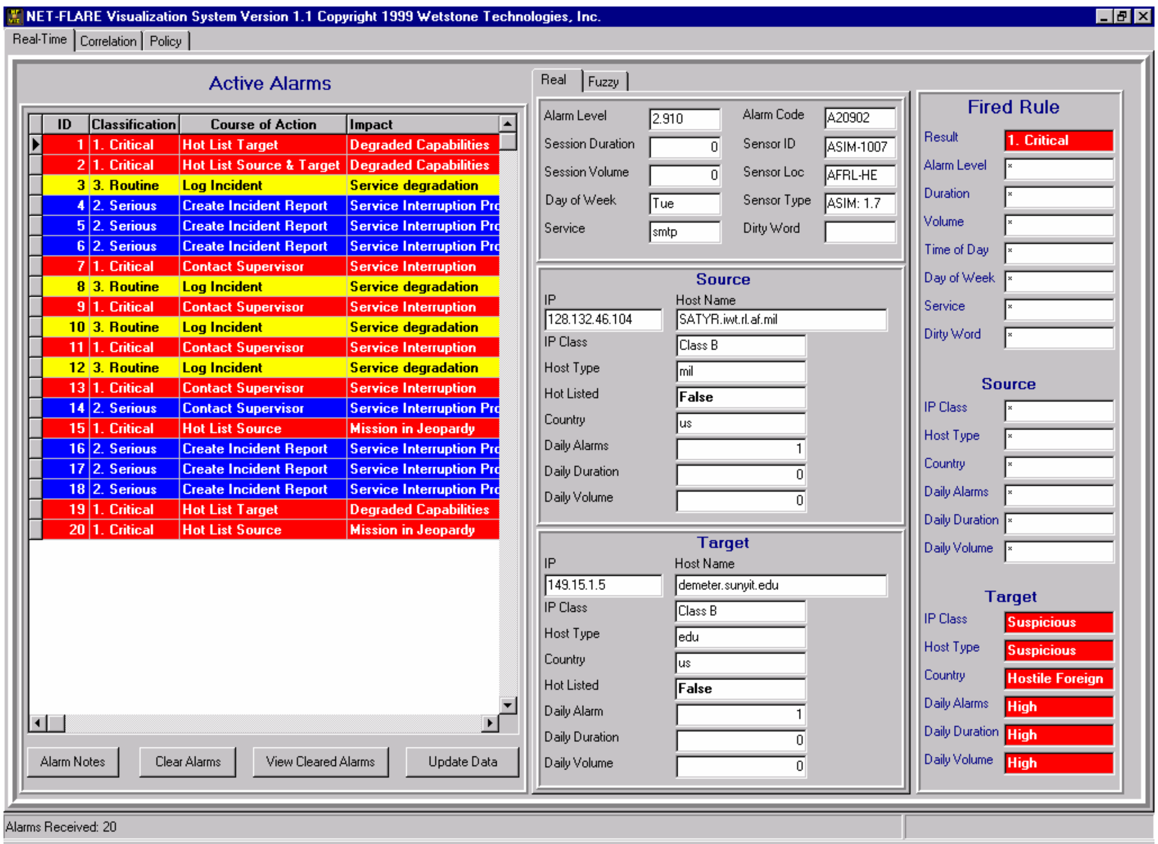


Figure 11 - NET-FLARE Visualization Engine

The correlation features of the visualization engine automatically correlate data that arrives from a particular source, is target toward a specific destination or originates from the same country. This provides the analyst with the ability to quickly examine results from multiple sensor events.

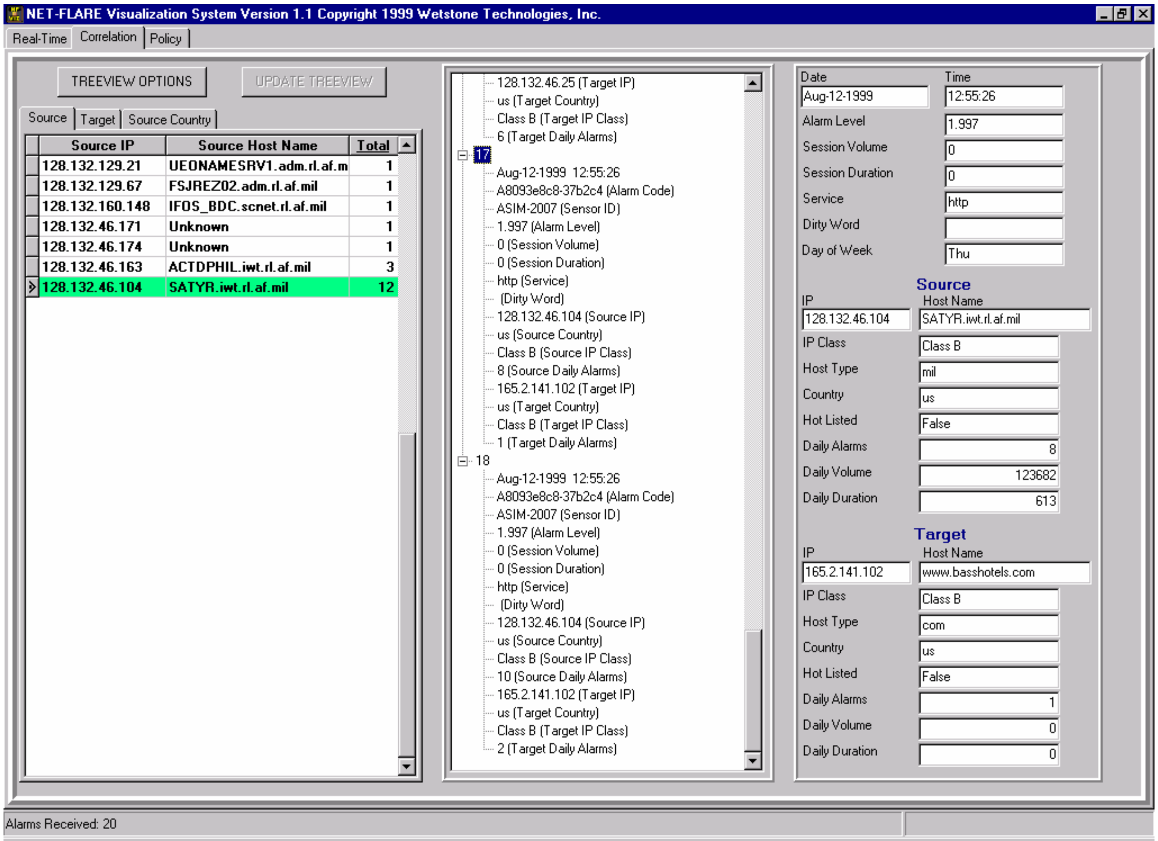


Figure 12 - NET-FLARE Correlation

4.9 Expert Technology Transfer

During this task WetStone staff worked on-site with the AFRL staff at Rome and their contractors to advance information protection capabilities into adjunct projects such as EPIC and ACTD/AIDE.

Under the direction of Dwayne Allain, Brian Spink and Mike Nassif, WetStone Technologies was directed to port the Network Monitoring and Assessment technology to Solaris and integrate it with the AIDE environment.

4.9.1 Scope

The IA:AIDE ACTD (AIDE) includes capabilities to monitor, correlate, and assess computer network intrusions. It accepts input from multiple sensors, including ASIM, JIDS and Network Radar. WetStone Technologies, Inc., under an Air Force contract, has developed a unique correlation and assessment system for ASIM data. This correlation module is called Network Monitoring and Assessment (NMA). The correlation that NMA can do on ASIM data can be further enhanced.

The purpose of this effort was to have WetStone Technologies, Inc., extend the current Network Monitoring and Assessment (NMA) Module to cover data available from ASIM and other, related sensors (e.g., those derived from the same base as ASIM); and provide assistance to PRC staff at the AFRL/Rome Site during the integration of the modified NMA.

4.9.2 NMA into IA:AIDE

WetStone Technologies worked with PRC staff at the AFRL Rome Site to determine what modifications needed to be made to the NMA to allow for compatibility with IA:AIDE on a Sun UltraSparc under the Solaris operating system. Our initial design for the modified system is shown in the following figure.

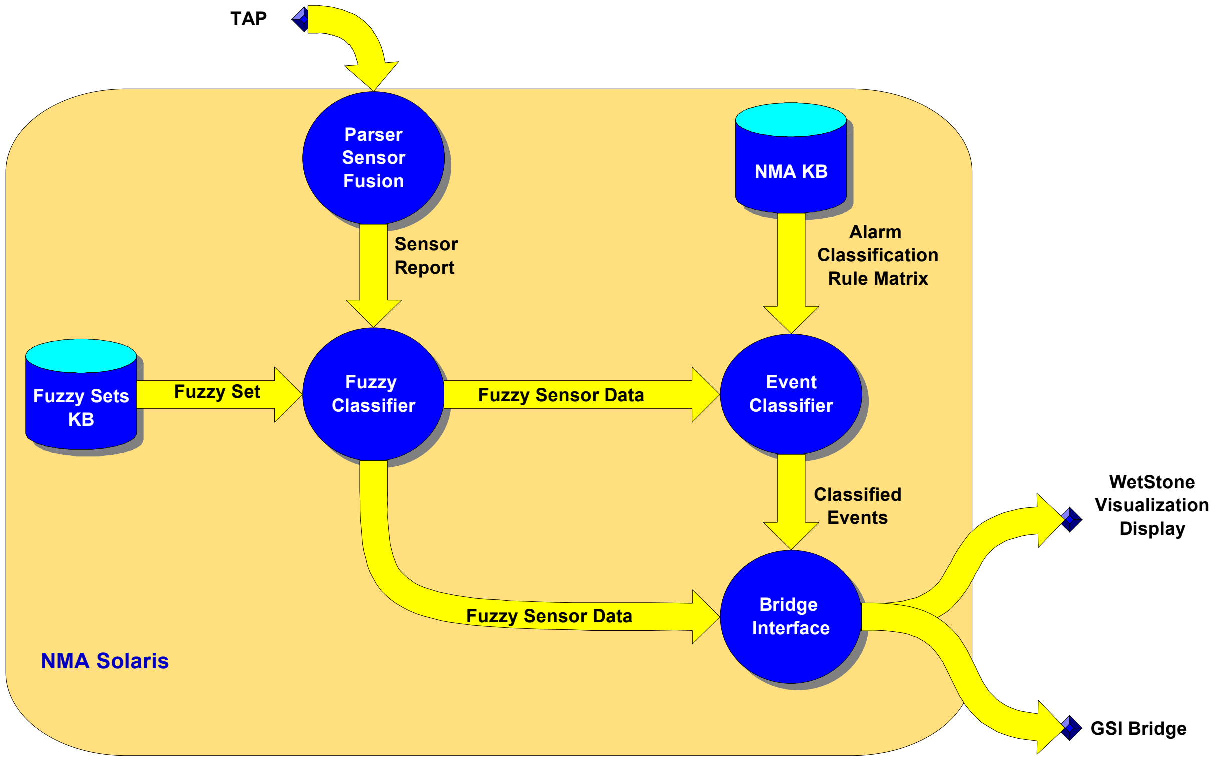


Figure 13 - NMA Solaris

4.9.2.1 Parser Sensor Fusion

This process receives and acknowledges requests from the AIDE Parser.

Requirements

1. This module will be developed as a process under the Sun Solaris OS.
2. TCP/IP Server Socket Interface will accept raw sensor data from the TAP.
3. Once the TAP transmits the NSE (TCP Send function), the Parser will shutdown and close the socket.
4. An internal data object NSE_OBJ will be created for each NSE received and then passed on to the Fuzzy Classifier Agent. Memory allocation responsibilities will be passed with the object.

4.9.2.2 Fuzzy Classifier

This module accepts requests from the Sensor Fusion Agent and applies the Fuzzy knowledgebase rules to each NSE.

Requirements

The module will be developed as a process under the Sun Solaris OS.

Each NSE field will be evaluated against the Fuzzy KB and assigned a Fuzzy value.
The Fuzzy values will be added to the NSE_OBJ and passed on to the Event Classifier.
Memory allocation responsibilities will be passed with the object to the Event Classifier.

4.9.2.3 Event Classifier

The Event Classifier accepts requests from the Fuzzy Classifier and applies the NMA KB rules to the NSE_OBJ.

Requirements

The module will be developed as a process under the Sun Solaris OS.
Each NSE_OBJ's Fuzzy values will be used to search the NMA KB to find a matching classification rule. Once found the classification rule and the corresponding recommended action will be updated in the NSE_OBJ.
The Fuzzy values will be added to the NSE_OBJ and passed on to the Bridge Interface.
Memory allocation responsibilities will be passed with the object to the Event Classifier.

4.9.2.4 Bridge Interface

The Bridge Interface accepts requests from the Event Classifier, and applies and formats the output for transmission via a TCP/IP socket to the WetStone Technologies' Visualization Environment and the GSI-Bridge.

Requirements

1. The module will be developed as a process under the Sun Solaris OS.
2. Each NSE_OBJ's Fuzzy values and Event Classification values formatted for transmission to the GSI-Bridge and the Visualization component
3. The content of the transmission will abide by the ACTD-AIDE standard for Normalized data.
4. The TCP/IP socket software will transmit the results
5. The NSE_OBJ will be logged.
6. The NSE_OBJ will be destructed and the memory released.

4.9.3 NMA (AIDE) Detailed Design

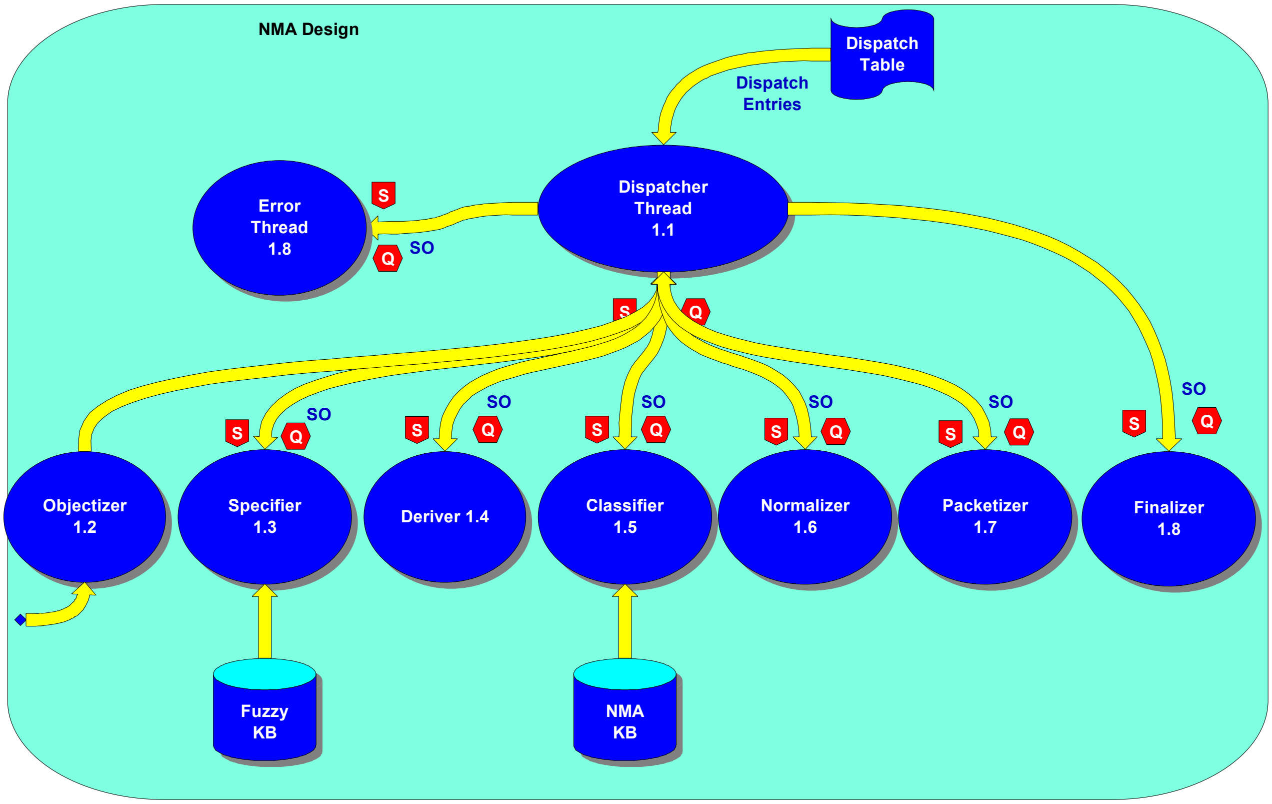


Figure 14 - NMA (AIDE) Detailed Design

Above we show the NMA detailed design. The NMA has 7 threaded processes that can be configured to interoperate based upon the configuration of the dispatch thread. The dispatch thread controls the operation based upon the state of the Sensor Object (SO) and the dispatch table.

4.9.3.1 General Thread Operation

Each NMA thread works on a simple principle. The thread waits for a single event (a semaphore) that alerts the thread that its queue needs attention. (Note: The semaphores that will be used are of the counting variety). The thread will process the SO's, based upon the requirements of the thread and update the state of the SO based upon the success or failure of the thread.

The general structure of the SO is as follows:

```
enum  SO_State {PARSED, FUZZY_SPECIFIED, FUZZY_CLASSIFIED,
FORWARDED, LOGGED, ERROR};

enum SO_SensorType {ASIM, JIDS, NETRADAR, TCPWRAPPERS};

class SO
{
     private:
          int  soID;
          SO_State state;
          SO_SensorType type;
          // private member functions
          …
          …
     public:

          // sensor data structure

          // public member functions
}
```

Objectizer

The process begins when the Objectizer thread receives an event or alarm from a connected sensor. The Objectizer (parser) creates an SO for each event received. As defined above, the parser object parses the alarm stream and extracts the individual data fields from the alarm data. The parser then builds the SO for the event received. Each SO object has a private member variable "state", the state variable is set to the "PARSED" state and is forwarded to the dispatcher's Queue and the Dispatchers Semaphore is called to notify the dispatcher that an SO object is ready for processing. In the current design the dispatcher will forward the SO on to the Fuzzy Specifier Thread based on the dispatch table configuration. This can be modified by changing the dispatch table in the future.

Specifier

This process begins when the Specifier receives a semaphore indicating a new SO has arrived in the input queue. The Specifier then processes the object and classifies each sensor data element to a Fuzzy values based on the Fuzzy Sets KB. Once the fuzzy specification process is completed, the SO state is set to FUZZY_SPECIFIED. The object is then placed in the Dispatch Queue and the Dispatch Semaphore is increased. The Specifier then goes back to waiting for a new SO object to arrive.

Normalizer

The Normalizer is responsible for converting the SO object into the normalized stream specified by the ACTD. The normalized data is transmitted to the Packetizer using the established TCP/IP socket.

Classifier

As with the Specifier, the Classifier waits for a new SO object to arrive. Once a new SO is available for processing, the Classifier searches the NMA KB for a matching rule based upon the fuzzy values contained in the SO. The NMA classifies the SO based on the rules in the NMA KB. Once completed the SO state is set to FUZZY_CLASSIFED and sent to the Dispatch Queue and the Dispatch Semaphore is increased.

Packetizer

When the Packetizer receives and SO object it will establish a communication session with the ACTD-AIDE Bridge code. The SO object will be converted into the normalized data stream specified by ACTD. Once normalized, the data will be transmitted to the Packetizer using the established TCP/IP socket. Once completed, the SO state is set to FORWARDED and sent to the Dispatch Queue and the Dispatch Semaphore is increased.

Finalizer

When the Logger receives a new SO, the SO is logged (appended) to the current day's log file. Once completed the SO is destroyed.

Error Thread

When the Error Thread receives a new SO the SO is logged (appended) tot he current days Error Log File. Once completed the SO is destroyed.

4.9.3.2 NMA/AIDE Implementation

The NMA module accepts real-time input from ASIM, JIDS and NetRadar sensors, filtered through corresponding bridges. NMA parses the data, derives additional data from the raw data, assigns fuzzy values to data fields, recommends a course of action, and produces normalized output. The output is fed to the Gensym system or TIS bridge. The NMA module can also display the details of each sensor message before it becomes normalized on a local screen.

Since the NMA interfaces with the bridges, it follows the configuration of the bridges' interface. Originally, the sensor bridges were providing output on a single socket, and opened and closed the socket for every message. The original configuration for NMA is presented in the top figure on the next page. The bridges' interface was changed to the socket stream concept, where each bridge opens a socket and keeps it open throughout the connection. This concept is used for both input to and output from NMA. The new NMA configuration is presented in the bottom figure.

The NMA Module was tested using real-time simultaneous sensor data from ASIM_2.0 and JIDS, and TIS and Gensym output.

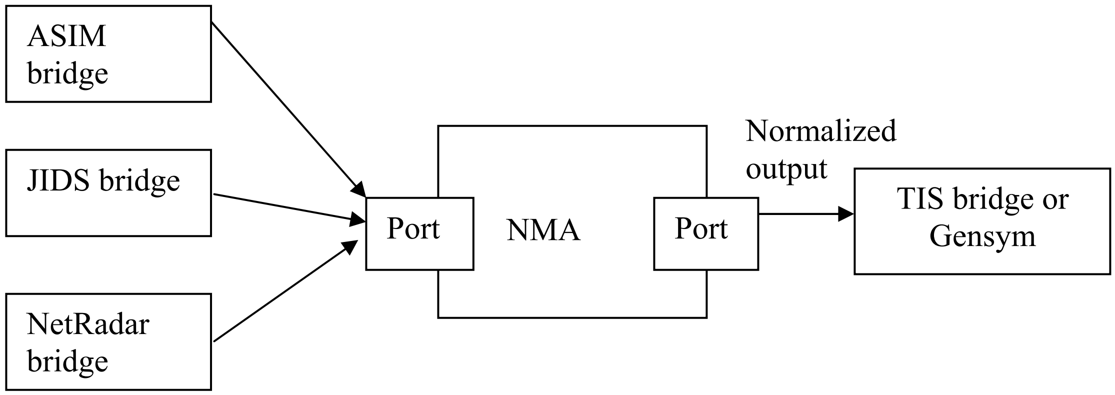


Figure 15 - Original Configuration of NMA Interface

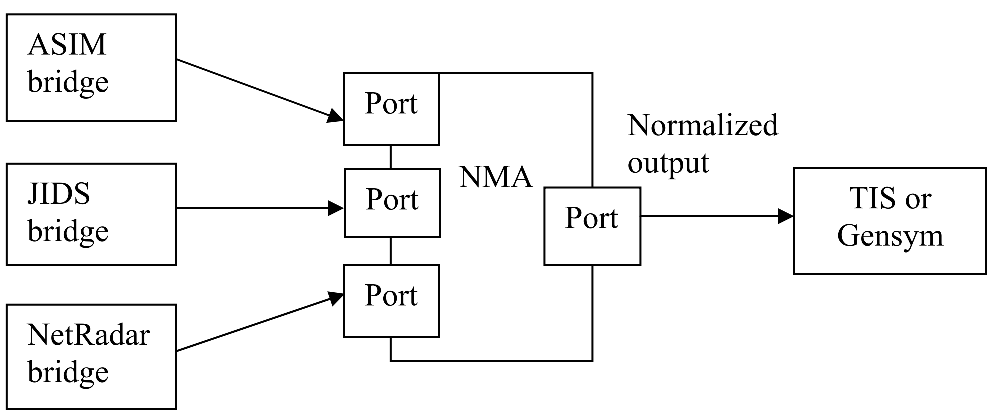


Figure 16 - Modified Configuration of NMA Interface

4.9.3.3 Modified NMA Structure

In this section we will cover individual NMA modules in more detail.

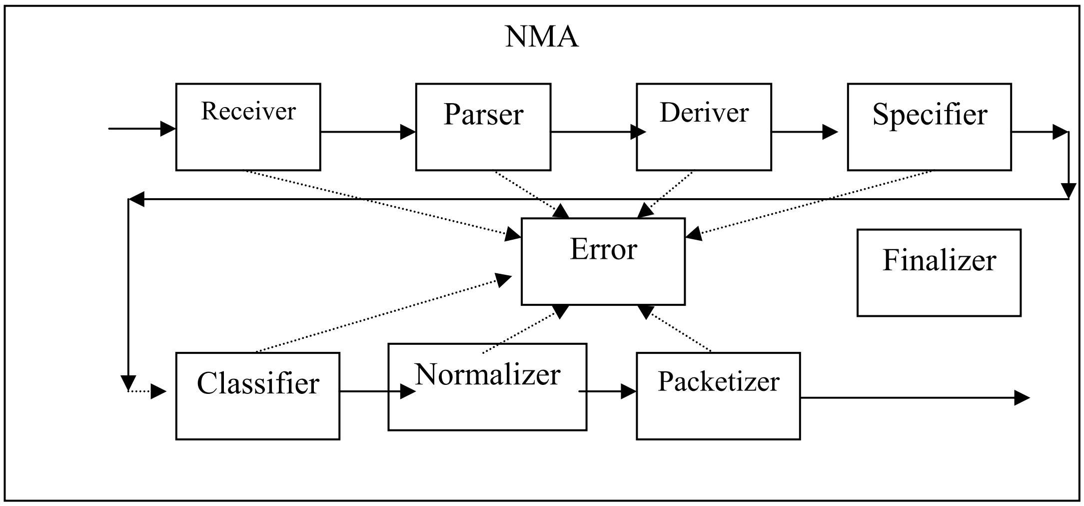


Figure 17 - Internal Box Diagram of NMA Module

The changes in the modules can be summarized as:

- The original *Receiver* and *Packetizer* modules have been updated to reflect the change in sensor input interface and TIS interface.
- All modules have been upgraded to process heartbeat messages.
- *Parser* and *Deriver* have been upgraded to process ASIM_20 data instead of ASIM_17 data.
- *Classifier* module has been added.

Heartbeat messages are messages transmitted by a sensor every minute to indicate that the sensor is alive. We added capability to process heartbeat messages received from ASIM and JIDS, and output a normalized string that indicates what sensor the heartbeat is from.

Receiver

The *Receiver* receives data from the sensors and thus provides the interface between NMA and sensors. Originally, the bridges' output opened and closed a socket for every sensor message, and *Receiver's* block diagram was as shown below. Performance concerns dictated that the overhead of opening and closing a socket for every message be eliminated and a socket opened and kept open throughout the connection. We changed the receiver to correspond to the new sensor output model, as presented on the following page.

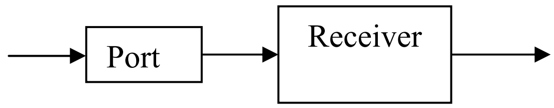


Figure 18 - Original Receiver Block Diagram

Opening and closing a socket for every message was automatically performing multiplexing of multiple concurrent sensor inputs, which the new model could not do, therefore we implemented multiplexing capability in the new model. The options included: opening a separate socket for every sensor and somehow multiplexing socket inputs; or writing a multiplexing module before messages are passed to the NMA input socket. The concurrent threading mechanism used for NMA architecture allowed us to receive different sensor input on different ports, and use the internal dispatching mechanism to multiplex the messages.

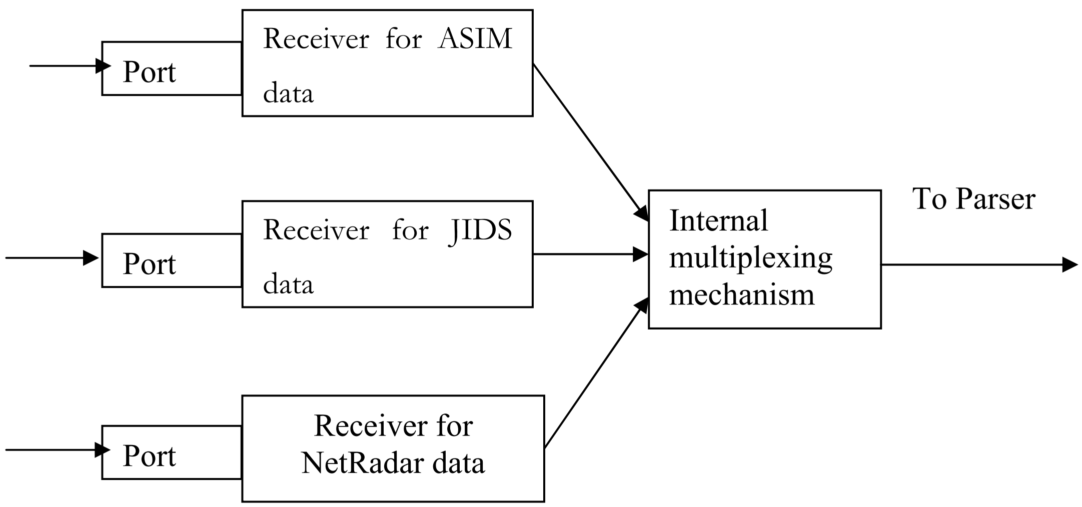


Figure 19 - New Receiver Block Diagram

Parser

The *Parser* module parses raw sensor data. Originally, *Parser* was designed for ASIM_17 data. We have upgraded it to parse ASIM_20 data, both connection and buzzword strings.

Deriver

The *Deriver* collects additional data based on the received raw sensor data. Some of the functions of *Deriver* include:

- Determine which host is source and which host is target. Source host is the host that is local; if both sender and receiver are local, the sender is the source.
- Perform DNS lookup to determine the host name corresponding to IP address, as well as host type.

- Check if a host is from the local domain, if a host is hotlisted, and determine which country the host is located in. This information is kept in separate files for easy editing without recompilation.
- Map port numbers into service names. This information is kept in a separate file for easy editing without recompilation.
- Recognize buzzwords recognized by ASIM, NetRadar and JIDS.
- Derive military time, day and date.
- Keep track of cumulative number of bytes transferred from each host, as well as alarm levels raised.

Specifier

The *Specifier* assigns fuzzy values to raw and derived data fields. We assigned fuzzy values that can be easily edited by system administrators without recompiling, because they are kept in a separate file. Fuzzy values are assigned to time of day, day of week, alarm level, session duration, session volume, session alarm, cumulative session volume, duration and alarms raised from suspect and target, service, buzzword, and host country and type.

Classifier

The *Classifier* module assigns recommended course of action (COA) for each message. This module uses a rules table to assign a recommendation based on various fields of the message. This table is editable by system administrators.

Each message field type is represented by one column in the table, but some sensors can have multiple message fields of the same type. We devised an algorithm to take this into account. For example, JIDS can have multiple BUZZWORD fields. We construct a separate "message" for each duplicate field, and check each of these "messages" against the rule table. Therefore, we get one recommended COA for each "message". Out of these COAs we assign the most critical one as the overall recommended COA for the original message.

Normalizer

The *Normalizer* produces a normalized output string in the format accepted by TIS and Gensym. This output string contains fuzzy values of raw sensor data.

Packetizer

The *Packetizer* is the module used to send data on the output socket, to the TIS bridge or Gensym. It has been upgraded to keep a socket open throughout the connection, instead of opening and closing for each sensor message.

Finalizer

The *Finalizer* is the "garbage collecting" module, that deletes messages from the NMA module after they have left the module.

Error

The *Error* module is used to process messages that could not be sent for some reason, either because of invalid format or unavailable output interface.

4.9.3.4 Java GUI Interface

We added a Java GUI interface which allows for visual manipulation of the data and display, and editing of the situational policy table used by the *Classifier* module

Our approach uses the Java Swing package to implement tables. The Swing package has JTable model for displaying and editing tables.

5.0 Summary

Our findings are that most information warfare technology advancements are in the form of point technologies that solve specific problems under specific conditions. Most new advancements tend to re-invent at least part of the wheel, causing progress to be slow and lethargic.

The approach developed under this effort to advance the state-of-the-art in decision support for the Information Warrior is quite different. Our position is that decision support is very closely tied to the day-to-day situation and mission that the Information Warrior is presented with. This means that development of decision support solutions, recommended course of actions (COAs), situation assessments and risk analysis cannot be addressed using "point solutions", because they would be obsolete before they were released. This hypothesis is not conjecture on our part, but rather is based on the direction we received from mission planners and Information Warriors in the field that we interviewed during this effort.

To support the Information Warrior's mission WetStone Technologies Inc. has developed an initial decision support capability under this effort that:

1. Supports multiple GOTS/COTS sensor inputs
2. Provides "real time" results
3. Is user configurable
4. Provides "At-a-Glance" visual identification of events
5. Automatically performs initial data mining for each event
6. Automatically fuses sensor's signature based detection capability with a situational policy for event filtering and assessment
7. Is hosted on an affordable PC platform

We have also validated our approach and the tool's potential via user feedback from several DOD agencies including AFMC NOSC, NSA and DISA.